Aflevering 3

Composite Design Pattern

Gruppe 4

I4SWD

16/04-2018

|  |  |  |
| --- | --- | --- |
| Navn: | Studienr.: | auID |
| Daniel Hansen | 201601915 | 565631 |
| Fatima Kodro | 201609565 | 553932 |
| Martin Andersen | 201605036 | 565118 |
| Søren Bech | 201604784 | 568435 |

Indhold

[Indledning 3](#_Toc511424285)

[Struktur og dynamik 3](#_Toc511424286)

[Typer af design 3](#_Toc511424287)

[Eksempel på anvendelse 4](#_Toc511424288)

[SOLID 4](#_Toc511424289)

[Diagrammer 5](#_Toc511424290)

[Eksempel på træstruktur 7](#_Toc511424291)

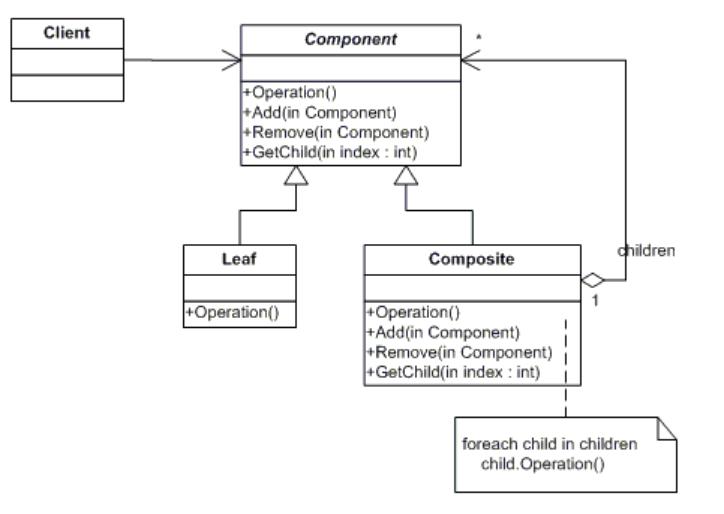
[Konklusion 7](#_Toc511424292)

# Indledning

Der er blevet valgt at arbejde med Composite pattern, som er ideel til at håndtere data i en træstruktur. Dette er begrundet af, at designet tager udgangspunkt i at individuelle objekter og kompositioner af objekter kan behandles ens. Composite pattern er derfor god at bruge hvis der ønskes at arbejde med komponenter, der er ”under” andre komponenter, dvs. komponenterne kan indeholde *children* (børn). Mulige træstrukturer kan være et hierarki i en virksomhed, hierarki i militæret eller et hierarki af komponenter som der anvendes i GUI-programmering. For at fremvise et eksempel på anvendelse af dette design pattern er der blevet opsat rang-system over militæret i flåden. Dette pattern er meget generelt og simpelt at implementere, men det kan være besværligt at tilføje nye metoder, idét det kræver at man ændrer eksisterende klasser.

# Struktur og dynamik

I Composite design pattern indgår der normalt 3 klasser: *Component, Composite* og *Leaf*. En klient (såsom et simpelt main program) kan derefter oprette instanser af klasserne. Component er *interfacet* til objekterne i kompositionen. Denne kan implementere default-behavior for metoder som er fælles for alle klasser, samt metoder som kan tilføje og fjerne børn-komponenter. Leaf er det primitive element: det har ingen børn og definerer metoder for det primitive objekt i kompositionen. Composite kan derimod gemme en række af børn og implementerer metoder som kan tilgå sine børn. Ved hjælp af rekursion vil det være meget nemt at traversere hele træstrukturen, da metoderne arves og *overrides* i det arvede klasser. Det er dog kun de relevante metoder for klasserne som overrides. Det generelle klassediagram kan ses på Figur 1.



Figur : Generel klassediagram indeholdende Component, Leaf, Composite og Client   
kilde: http://www.dofactory.com/net/composite-design-pattern

## Typer af design

Der findes to typer af design varianter for designet. Den beskrevne type er designet for *ensartethed*. Dette betyder at metoder som omhandler børn-komponenter er defineret i Component klassen. Dette betyder at Leaf og Composite er ens. Konsekvensen deraf er, at klienter kan foretage børn-relaterede operationer på Leaf objekter, som ikke burde være muligt. Dog kan man implementere en default-behavior som blot kaster en exception hvis den pågældende metode bliver kaldt på en metode som ikke er implementeret i Leaf (eller Composite).

Den anden type er designet for *typesikkerhed*. Dette er netop hvor man opdeler metoderne i henholdsvis Leaf og Composite og kun definerer fællesoperationer i Component. Dette garanterer typesikkerheden, men deraf skal klienten behandle objekterne forskelligt.

# Eksempel på anvendelse

Som eksempel er der valgt at arbejde med et rang-system over militæret i flåden, hvor et element kan defineres som at være en del af et andet element eller underordnede. Eksemplet består af Leaves og Composites, hvor Leaf i dette tilfælde er den rang, som ikke kan have nogle underordnede. Denne rang er ”Sailor”. Composites er derimod dem, der har mulighed for at have en eller flere underordnede. Da der er flere funktioner der kan bruges i både Leaves og Composite, samt er der funktioner ud over disse, bruges en component klasse, som beskriver alle funktioner, både dem der er fælles, samt dem, der virker for Leaves og Composites. Dette betyder at designet for ensartethed er anvendt.

Med Composite pattern er det nemt at traversere alle elementer i træstrukturen, samt er det muligt at traversere over specifikke dele, f.eks. en kaptajn og alle under ham. Det er muligt at implementere en metode som kan tilgå en komponents forældre i det rekursive træ for at tilgå sin overordnede. Dette kan give mening hvis man vil vide hvem den overordnede er til et bestemt barn, hvis man skal udføre bestemte metoder herpå.

## SOLID

Overvejelser omkring SOLID beskrives som følgende:

* **S – Single responsibility princippet**

er ikke blevet brugt, da det giver mest mening at have en overordnet klasse, Component, for alle Composite og Leaf. Dette medfører at Component-klassen har flere forskellige kommandoer, der kan udføres og derfor er der ikke single responsibility.

* **O – Open-closed princippet**

er ikke brugt, da hvis der skal ændres noget i enten Leaf- eller Composite klassen, skal dette også ændres i Component klassen. Dette er nødvendigt, da der netop er taget udgangspunkt i ensformighed frem for typesikkerhed. Dog hvis typesikkerhed-designet blev brugt, ville mindst en klasse stadig blive lavet om, medmindre man tilføjer en ny klasse som implementere igen eller definere nye metoder.

* **L – Liskov substitution princippet**

bruges i eksemplet, da det er muligt at substituere alle Leaves og Composites. Dette kan ske hvis man vælger at implementere en metode som kan degradere en person fra en Composite til en Leaf.

* **I – Interface segregation** **princippet**

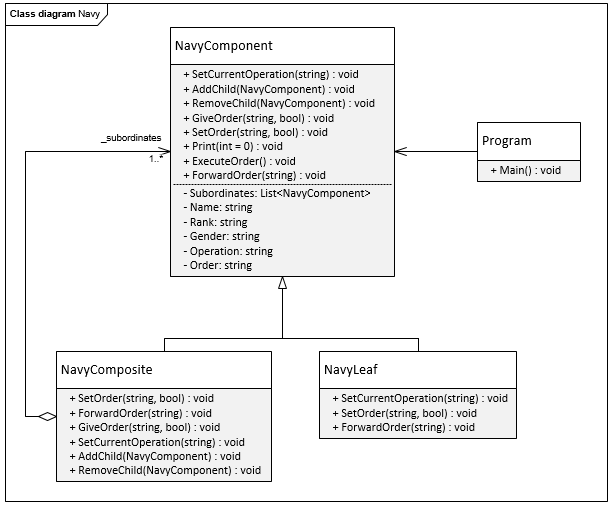
bruges ikke, da der ikke var brug for at lave interfaces til klasserne.

* **D - Dependency inversion princippet**

bruges, da der er implementeret en Component abstract klasse, som Leaf og Composite er afhængig af, som er konkrete klasser. Dette betyder at Leaf og Composite ikke er afhængige af hinanden.

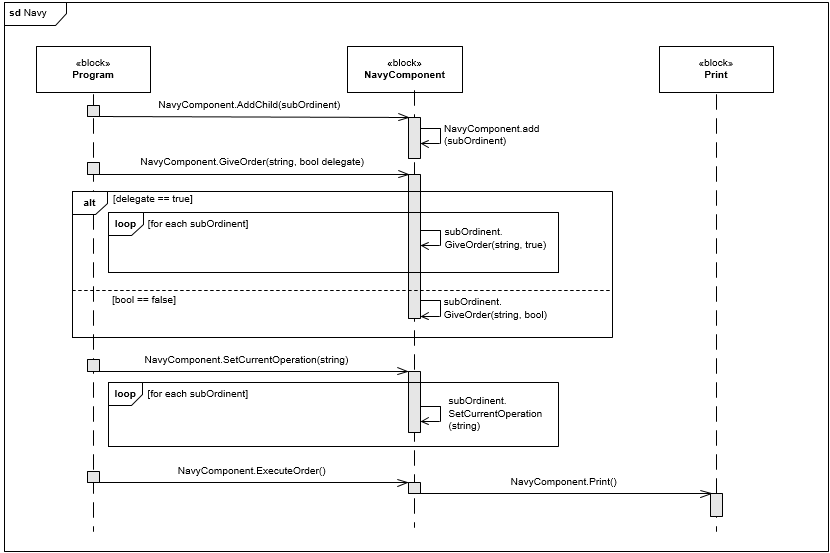
## Diagrammer

På Figur 2, ses et klassediagram over Navy, hvor det ses at NavyComposite og NavyLeaf arver fra NavyComponent. Det er som beskrevet tidligere, fordi Component er en overordnet klasse over de andre. NavyLeaf beskriver den rang som er nederst i systemet, dvs. de kan ikke have nogen underordnede, mens NavyComposite kan indeholde en liste af NavyComponents. Derfor har de kun mulighed for at få ordrer og operationer. Composites beskriver de overordnede rang, så derfor har de flere kommandoer, fx at give ordrer og operationer ud til andre. Alle operationer er implementeret med en default-behavior i NavyComponent som blot kaster en exception. Dette betyder, at hvis f.eks. Leaf prøver at kalde en funktion som den ikke selv har implementeret vil en exception blive kastet.



Figur : Klassediagram over Navy

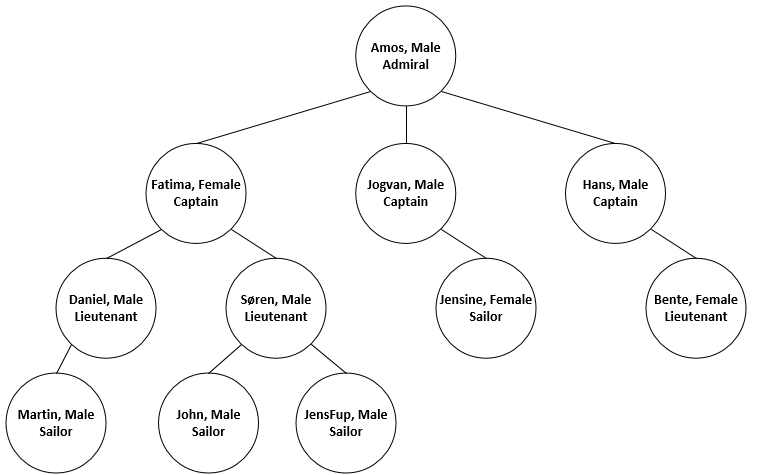
På Figur 3 ses et sekvensdiagram over Navy. Her vises et scenarie af hvilke funktioner, der bliver kaldt i programmet. Der bruges kun NavyComponent, da den repræsenterer både Leaf og Composite. Der vises til at starte med, at der bliver tilføjet en underordnede til en component vha. AddChild-metoden. Herefter gives der en ordrer til denne underordnede. Når der gives en ordre, kan parameteren ”delegate”, som er en bool, sættes til enten true eller false. Hvis den sættes til true, vil ordren blive sendt videre ned i hierarkiet, dvs. den bliver sendt ned til alle Leaves for den valgte Composite. Hvis ”delegate” sættes til false, er det kun de personer der er lige under den valgte Composite der får ordren. Herefter bliver der kaldt SetCurrentOperation(). Når dette bliver kaldt, bliver der udført en operation til en Composite og alle dens underordnede. Til sidst bliver der kaldt ExecuteOrder, som udfører en Components ordre, hvis der er tildelt en til personen.



Figur : Sekvensdiagram over Navy

## Eksempel på træstruktur

Den konkrete træstruktur der blevet brugt i kode-eksemplet kan ses på Figur 4, hvor der i hver knude er beskrevet navn, køn og rang for hver person. Dette er forsøgt udskrevet på en anskuelig måde i koden.



Figur : Træstruktur over Navy

# Konklusion

Composite er godt at bruge, hvis man har en træstruktur som man vil kunne arbejde med. Her kan man behandle komponerede objekter som individuelle objekter. Det er på denne måde nemt for klienten at udføre operationer på strukturen. Det er også ret nemt at tilføje nye typer af komponenter.

Composite pattern har sine fordele men kan blive for generel. Hvis man har flere forskellige implementeringer af Component, er det svært at begrænse til én type af Component i en Composite. Skal dette være muligt, skal der ske run-time tjek i koden.

Composite pattern kan sammenlignes med Decorator pattern. Implementeringen af dem kan være rimelig ens, men formålet er forskelligt. Decorater skal dekorere en enkelt komponent, hvor Composite skal gruppere komponenter som en helhed. Decorater har til formål at tilføje funktionalitet, hvor Composite vil samle komponenter. Desuden tager Decorater også stor udgangspunkt i Single Responsibility Princippet, idét at funktionaliteten deles ud på forskellige klasser.